Computability and randomness

SET07106 Mathematics for Software Engineering

School of Computing
Edinburgh Napier University
Module Leader: Uta Priss

2010

Outline

Computability

Randomness

Cellular automata

How fast is a program?

Is it possible to know how long it will take for a program to compute something?

How fast is a program?

Is it possible to know how long it will take for a program to compute something?

One can often prove that the complexity of a program behaves like a mathematical function.

Complexity of programs

notation	meaning	example
O(1)	constant	check whether number is odd or even
O(log n)	logarithmic	finding item in sorted list
O(n)	linear	finding item in unsorted list
$O(n^c)$	polynomial	sorting, matrix multiplication, parsing
$O(c^n)$	exponential	travelling salesman problem (exact solution)

The desired complexity is **polynomial** or faster.

Problems with higher complexity cannot practically be solved for larger values of n. (Example: encryption)

Halting

This program finishes really quickly:

```
print "Hello World"
```

This program does not **halt**:

```
while 1:
      print "Hello World"
```

It contains an infinite loop.

Turing's uncomputability

Halting problem:

Is it possible to know whether a program halts by just examining the code?

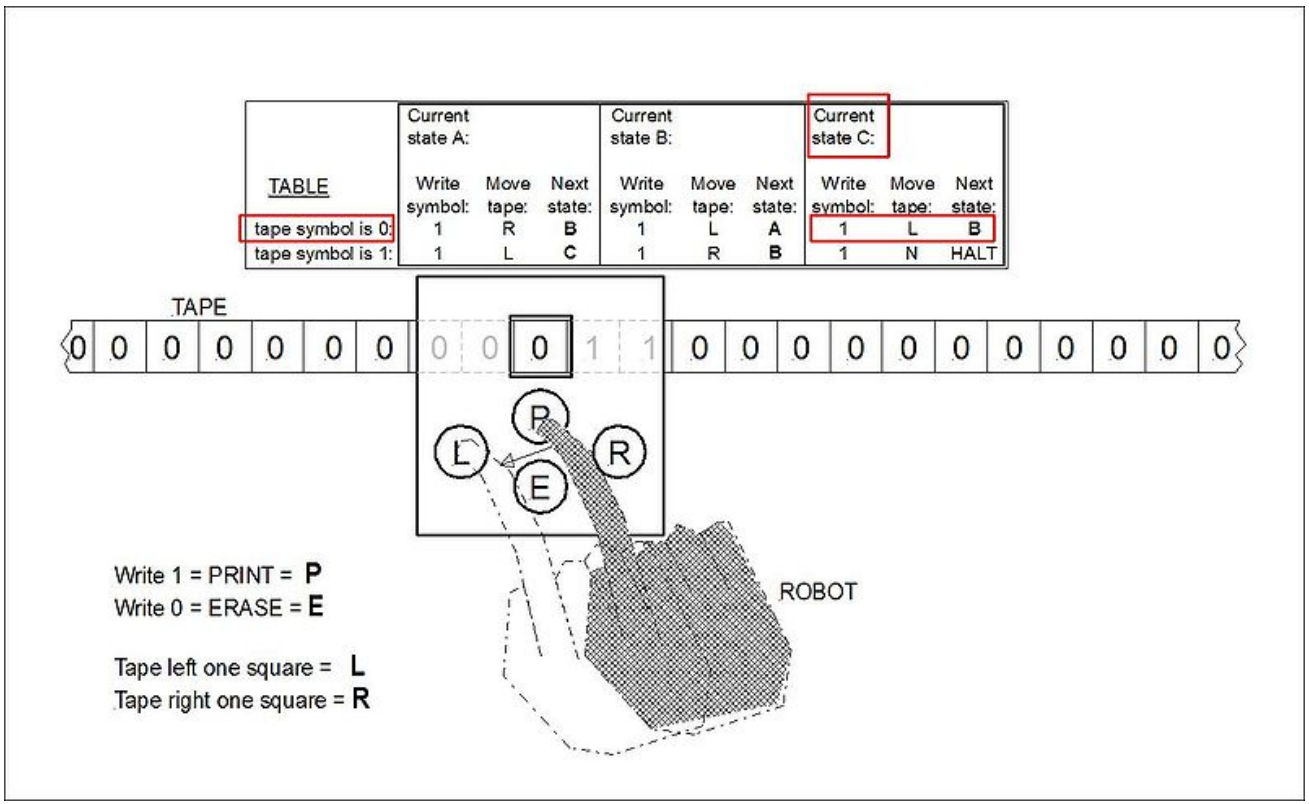
Turing's uncomputability

Halting problem:

Is it possible to know whether a program halts by just examining the code?

Turing proved in 1936 that this problem is **undecidable** for Turing machines.

A Turing machine



(Source: user Wvbailey, Wikipedia)

Turing machines

Every computer is equivalent to a Turing machine.

$\Longrightarrow$ Every computer can compute the same problems.
(However: some computers are faster than others.)

Turing machines

Every computer is equivalent to a Turing machine.

$\Longrightarrow$ Every computer can compute the same problems.
(However: some computers are faster than others.)

Quantum computers may be more powerful than Turing machines.
But so far it is not clear whether they can be built on a larger scale.

Turing completeness of programming languages

Every programming language can compute the same problems.

If it is not possible to write a certain program so that it halts in Java, it cannot be written in C, PHP, Python, etc either.

Turing completeness of programming languages

Every programming language can compute the same problems.

If it is not possible to write a certain program so that it halts in Java, it cannot be written in C, PHP, Python, etc either.

Of course, this does not mean that every programmer can write every program!

Non-computable problems

Proving that a problem is non-computable does not mean that a solution cannot be computed in some cases, but that it is impossible to construct a solution that works for all inputs.

Non-computable problems

Proving that a problem is non-computable does not mean that a solution cannot be computed in some cases, but that it is impossible to construct a solution that works for all inputs.

For example:
It is not clear for all programs whether they terminate.
But everyday software is usually written so that it does terminate.

Gödel's incompleteness

Every system that is reasonably complex contains statements that cannot be proven (or disproven) from the axioms of the system.

Example:

It is not possible to prove all laws about basic arithmetics from the axioms!

How much do we know about mathematics?

How many mathematical problems are there?

How many mathematical problems have been solved so far?

Chaitin's randomness

Almost every mathematical statement is true for no reason and cannot be proven from other axioms.

Most mathematical statements are

unconnected, random, and unprovable.

Chaitin's program-size complexity

The complexity of some problem can be measured by the size of the smallest program that calculates it.

Program-size complexity is uncomputable.

If a statement is more complex than a set of axioms
$\Rightarrow$ the statement cannot be proven from the axioms.

Chaitin: unprovable statements

1, 2, 3, 4, 5, 6, ... 17, 18, 19, ... ?

The first uninteresting integer.

Chaitin: unprovable statements

1, 2, 3, 4, 5, 6, ... 17, 18, 19, ... ?

The first uninteresting integer.

It cannot be proven whether a number is interesting.

Experimental data versus proof

There are many examples of mathematical statements that appear to be true based on experimental data, but cannot be proven.

- Fermat's Last Theorem (no positive integers can satisfy $a^n + b^n = c^n$): was solved after 358 years.
- Four colour theorem (whether a map can always be coloured with 4 colours so that adjacent countries have different colours): proven with a computer.
- Riemann Hypothesis (Distribution of the zeros of the Riemann zeta-function): remains unsolved.

Extension and intension

Experimental results are extensional.

Proofs using axioms and inference rules are intensional.

Non-deterministic algorithms

Algorithms that use randomness can sometimes compute uncomputable questions with high probability of being true!

Stephen Wolfram: A new kind of science

Cellular automata:

- simple algorithms generate complex behaviour
- nature is based on simple algorithms
- all such algorithms are of the same complexity
- algorithms are more powerful than equations
- finding simple rules instead of finding explanations

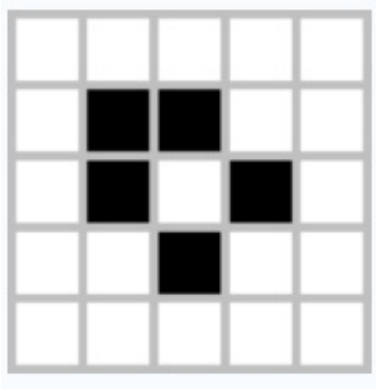
Examples

- a jaguar's spots
- bird flight patterns
- the stock market
- the growths of leaves

Conway's Game of Life

Each cell has 8 neighbours.

- Any live cell with fewer than 2 live neighbours dies.
- Any live cell with more than 3 live neighbours dies.
- Any live cell with 2 or 3 live neighbours lives on to the next generation.
- Any dead cell with exactly 3 live neighbours becomes a live cell.



Game of Life

A glider is a pattern that travels across the board in Conway's Game of Life.

It is possible to construct logic gates such as AND, OR and NOT using gliders.

The Game of Life is Turing complete: it is as powerful as any computer with unlimited memory and no time constraints.